Aalto University
School of Science

CS-E4530 Computational Complexity Theory

Lecture 5: Boolean Logic

Aalto University
School of Science
Department of Computer Science

Spring 2017

Agenda

Boolean logic:

- syntax
- semantics
- normal forms
- satisfiability and validity
- Boolean functions and expressions
- Boolean circuits

(C. Papadimitriou: *Computational Complexity*, Chapter 4)

Motivation

- Logic is a fundamental representation language in many areas of computer science:
  digital circuit design, programming language semantics, specification and verification, constraint programming, logic programming, databases, artificial intelligence, knowledge representation, machine learning, ...
- Logic involves interesting computational problems.
- In computational complexity theory:
  Computational problems from logic can be shown to be "universal" at various levels of computational difficulty.
  This leads to important insights and techniques for reasoning about computational complexity issues.

1. Syntax

- The syntax of Boolean logic (i.e. the set of well-formed Boolean expressions) is based on the following symbols:
  – Boolean *variables* (or *atoms*): $X = \{x_1, x_2, \ldots\}$.
  – Boolean *connectives*: $\vee$ (or), $\wedge$ (and), and $\neg$ (negation).
- The set of Boolean expressions (formulas) is inductively defined as follows:
  - all Boolean variables are Boolean expressions,
  - if ϕ_1 and ϕ_2 are Boolean expressions, then so are $\neg\phi_1$, $(\phi_1 \wedge \phi_2)$, and $(\phi_1 \vee \phi_2)$, and
  - nothing else is a Boolean expression.
- When x_i is a Boolean variable, expressions of the form x_i and $\neg x_i$ are called *literals*.

Example

$((x_1 \vee x_2) \wedge \neg x_3)$ is a Boolean expression but $((x_1 \vee x_2)\neg x_3)$ is not.

Some notational conventions

- Simplified notation: $(((x_1 \vee \neg x_3) \vee x_2) \vee (x_4 \vee (x_2 \vee x_5)))$ is written as $x_1 \vee \neg x_3 \vee x_2 \vee x_4 \vee x_2 \vee x_5$ or $x_1 \vee \neg x_3 \vee x_2 \vee x_4 \vee x_5$.
- Disjunctions and conjunctions involving n members:
  - $\bigvee_{i=1}^{n} \varphi_i$ stands for $\varphi_1 \vee \cdots \vee \varphi_n$.
  - $\bigwedge_{i=1}^{n} \varphi_i$ stands for $\varphi_1 \wedge \cdots \wedge \varphi_n$.
- Other connectives can be defined as shorthands:
  - An implication $\phi_1 \rightarrow \phi_2$ stands for $\neg\phi_1 \vee \phi_2$.
  - An equivalence $\phi_1 \leftrightarrow \phi_2$ stands for $(\neg\phi_1 \vee \phi_2) \wedge (\neg\phi_2 \vee \phi_1)$.

2. Semantics

How to interpret Boolean expressions?

- Boolean expressions are propositions that are either true or false. They speak about a world where certain atomic propositions (Boolean variables) are either true or false.
  Such a truth assignment induces then a truth value for any Boolean expression built from the Boolean variables.
- A *truth assignment* T is a mapping from a finite subset $X' \subseteq X$ to the set of truth values $\{\mathbf{true}, \mathbf{false}\}$.
- Let $X(\phi)$ be the set of Boolean *variables appearing in* ϕ.
- A truth assignment $T : X' \to \{\mathbf{true}, \mathbf{false}\}$ is *appropriate* to ϕ if $X(\phi) \subseteq X'$.

The Satisfaction Relation

- Let a truth assignment $T : X' \to \{\mathbf{true}, \mathbf{false}\}$ be appropriate to ϕ, i.e., $X(\phi) \subseteq X'$.
- $T \models \phi$ (T *satisfies* ϕ) is defined inductively as follows:
  - If ϕ is a variable from X', then $T \models \phi$ iff $T(\phi) = \mathbf{true}$.
  - If $\phi = \neg\phi_1$, then $T \models \phi$ iff $T \not\models \phi_1$.
  - If $\phi = \phi_1 \wedge \phi_2$, then $T \models \phi$ iff $T \models \phi_1$ and $T \models \phi_2$.
  - If $\phi = \phi_1 \vee \phi_2$, then $T \models \phi$ iff $T \models \phi_1$ or $T \models \phi_2$.

Example

Let $T = \{x_1 \mapsto \mathbf{true}, x_2 \mapsto \mathbf{false}\}$.
Then $T \models x_1 \vee x_2$ but $T \not\models (x_1 \vee \neg x_2) \wedge (\neg x_1 \wedge x_2)$.

Logical equivalence

- Expressions ϕ_1 and ϕ_2 are logically *equivalent* ($\phi_1 \iff \phi_2$) iff for all truth assignments T appropriate to both of them,

$$T \models \phi_1 \text{ iff } T \models \phi_2.$$

Example

Let ϕ, ϕ_1, and ϕ_2 be expressions. We have, e.g., the following:

- $(\phi_1 \vee \phi_2) \iff (\phi_2 \vee \phi_1)$
- $((\phi_1 \wedge \phi_2) \wedge \phi_3) \iff (\phi_1 \wedge (\phi_2 \wedge \phi_3))$
- $\neg\neg\phi \iff \phi$
- $((\phi_1 \wedge \phi_2) \vee \phi_3) \iff ((\phi_1 \vee \phi_3) \wedge (\phi_2 \vee \phi_3))$
- $\neg(\phi_1 \wedge \phi_2) \iff (\neg\phi_1 \vee \neg\phi_2)$
- $\neg(\phi_1 \vee \phi_2) \iff (\neg\phi_1 \wedge \neg\phi_2)$
- $(\phi_1 \vee \phi_1) \iff \phi_1$

3. Normal Forms

- The most frequently used normal forms for Boolean expressions are conjunctive and disjuntive normal forms (CNF/DNF).
- These forms are defined by
  CNF: $(l_{1,1} \vee \cdots \vee l_{1,n_1}) \wedge \cdots \wedge (l_{m,1} \vee \cdots \vee l_{m,n_m})$
  DNF: $(l_{1,1} \wedge \cdots \wedge l_{1,n_1}) \vee \cdots \vee (l_{m,1} \wedge \cdots \wedge l_{m,n_m})$
  where each l_{ij} is a literal (Boolean variable or its negation).
- A disjunction $l_1 \vee \cdots \vee l_n$ of literals is called a *clause*.
- A conjunction $l_1 \wedge \cdots \wedge l_n$ of literals is called an *implicant*.
- We can assume that normal forms do not have repeated clauses/implicants or repeated literals in clauses/implicants.

Example

$(\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3 \vee x_4) \wedge (\neg x_4)$ is in CNF.
$(\neg x_1 \wedge x_2) \vee (\neg x_2 \wedge x_3 \wedge x_4) \vee (\neg x_4)$ is in DNF.
$(\neg x_1 \vee \neg x_1 \vee x_2) \iff (\neg x_1 \vee x_2)$.

CNF/DNF Transformation

Theorem

Every Boolean expression is equivalent to one in conjunctive (disjunctive) normal form.

Proof by construction: any Boolean expression can be transformed into CNF/DNF as follows.

- First, remove $\leftrightarrow$ and $\rightarrow$:

$$\alpha \leftrightarrow \beta \quad \rightsquigarrow \quad (\neg\alpha \vee \beta) \wedge (\neg\beta \vee \alpha) \qquad (1)$$

$$\alpha \rightarrow \beta \quad \rightsquigarrow \quad \neg\alpha \vee \beta \qquad (2)$$

- Then, push negations in front of Boolean variables:

$$\neg\neg\alpha \quad \rightsquigarrow \quad \alpha \qquad (3)$$

$$\neg(\alpha \vee \beta) \quad \rightsquigarrow \quad \neg\alpha \wedge \neg\beta \qquad (4)$$

$$\neg(\alpha \wedge \beta) \quad \rightsquigarrow \quad \neg\alpha \vee \neg\beta \qquad (5)$$

☞ The result is a mixed conjunction and disjunction of literals.

CNF/DNF Transformation—cont'd

The next phase depends on the normal form being pursued:

- For the CNF, move $\wedge$ connectives outside $\vee$ connectives:
$$\alpha \vee (\beta \wedge \gamma) \quad \rightsquigarrow \quad (\alpha \vee \beta) \wedge (\alpha \vee \gamma) \qquad (6)$$
$$(\alpha \wedge \beta) \vee \gamma \quad \rightsquigarrow \quad (\alpha \vee \gamma) \wedge (\beta \vee \gamma) \qquad (7)$$
- For the DNF, move $\vee$ connectives outside $\wedge$ connectives:
$$\alpha \wedge (\beta \vee \gamma) \quad \rightsquigarrow \quad (\alpha \wedge \beta) \vee (\alpha \wedge \gamma) \qquad (8)$$
$$(\alpha \vee \beta) \wedge \gamma \quad \rightsquigarrow \quad (\alpha \wedge \gamma) \vee (\beta \wedge \gamma) \qquad (9)$$

Note: Normal forms can be exponentially bigger than the original expression in the worst case.

CNF/DNF Transformation—cont'd

Example

Transform $(x_1 \vee x_2) \rightarrow (x_2 \leftrightarrow x_3)$ into CNF.

$$
\begin{aligned}
& (x_1 \vee x_2) \rightarrow (x_2 \leftrightarrow x_3) && (2)\\
\iff & \neg(x_1 \vee x_2) \vee (x_2 \leftrightarrow x_3) && (1)\\
\iff & \neg(x_1 \vee x_2) \vee ((\neg x_2 \vee x_3) \wedge (\neg x_3 \vee x_2)) && (4)\\
\iff & (\neg x_1 \wedge \neg x_2) \vee ((\neg x_2 \vee x_3) \wedge (\neg x_3 \vee x_2)) && (7)\\
\iff & (\neg x_1 \vee ((\neg x_2 \vee x_3) \wedge (\neg x_3 \vee x_2))) \wedge (\neg x_2 \vee ((\neg x_2 \vee x_3) \wedge (\neg x_3 \vee x_2))) && (6)\\
\iff & ((\neg x_1 \vee (\neg x_2 \vee x_3)) \wedge (\neg x_1 \vee (\neg x_3 \vee x_2))) \wedge && \\
& (\neg x_2 \vee ((\neg x_2 \vee x_3) \wedge (\neg x_3 \vee x_2))) && (6)\\
\iff & ((\neg x_1 \vee (\neg x_2 \vee x_3)) \wedge (\neg x_1 \vee (\neg x_3 \vee x_2))) \wedge && \\
& ((\neg x_2 \vee (\neg x_2 \vee x_3)) \wedge (\neg x_2 \vee (\neg x_3 \vee x_2))) && \text{(Simplifica}\\
\iff & (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_3 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3 \vee x_2) &&
\end{aligned}
$$

4. Satisfiability and Validity

- A Boolean expression ϕ is *satisfiable* iff
  there is a truth assignment T appropriate to it such that $T \models \phi$.
- A Boolean expression ϕ is *valid*/*tautology* (denoted by $\models \phi$) iff
  for every truth assignment T appropriate to it, $T \models \phi$.
- The interconnection of satisfiability and validity:

  $$\phi \text{ is valid } (\models \phi) \text{ iff } \neg\phi \text{ is unsatisfiable.}$$

- Moreover, for any Boolean expressions ψ_1 and ψ_2,

  $$\psi_1 \iff \psi_2 \text{ iff } \models \psi_1 \leftrightarrow \psi_2 \text{ iff } \neg(\psi_1 \leftrightarrow \psi_2) \text{ is unsatisfiable.}$$

Example

- $(x_1 \vee \neg x_2) \wedge \neg x_1$ is satisfiable
- $(x_1 \vee \neg x_2) \wedge \neg x_1 \wedge x_2$ is unsatisfiable
- $(\neg x_1 \wedge x_2) \vee x_1 \vee \neg x_2$ is valid

☞ Satisfiability forms a fundamental computational problem.

Definition (The Satisfiability Problem)

SAT:
INSTANCE: an expression φ in CNF
QUESTION: is φ satisfiable?

- SAT can be solved in $O(n^2 2^n)$ time (e.g., by the truth table method).
- SAT $\in \mathbf{NP}$:

A nondeterministic Turing machine for $\varphi \in$ SAT:
for all variables x in φ **do**
    choose nondeterministically: $T(x) := \mathbf{true}$ or $T(x) := \mathbf{false}$;
if $T \models \varphi$ **then** return "yes" **else** return "no"

- But whether SAT $\in \mathbf{P}$ holds is an open question!

Horn clauses and HORNSAT

- An interesting special case of SAT concerns *Horn clauses*, i.e., clauses (disjunction of literals) with *at most one positive literal*.

Example

The clauses $(\neg x_1 \vee x_2 \vee \neg x_3)$, $(\neg x_1 \vee \neg x_3)$, and (x_2) are Horn clauses but $(\neg x_1 \vee x_2 \vee x_3)$ is not.

- A Horn clause $(\neg x_1 \vee x_2 \vee \neg x_3)$ with a positive literal is called an *implication* and can be written as $(x_1 \wedge x_3) \rightarrow x_2$ (or $\rightarrow x_2$ when there are no negative literals).

Definition

HORNSAT:
INSTANCE: a conjunction φ of Horn clauses.
QUESTION: is φ satisfiable?

A Polynomial Time Algorithm for HORNSAT

Algorithm $hornsat(S)$
/* Determines whether $S \in$ HORNSAT */
$T := \emptyset$ /* T is the set of true atoms */
repeat
    if there is an implication $(x_1 \wedge x_2 \wedge \cdots \wedge x_n) \rightarrow y$ in S
        such that $\{x_1, \ldots, x_n\} \subseteq T$ but $y \notin T$ **then**
        $T := T \cup \{y\}$
until T does not change
if for all purely negative clauses $\neg x_1 \vee \cdots \vee \neg x_n$ in S,
    there is some literal $\neg x_i$ such that $x_i \notin T$ **then**
    return S is satisfiable
else return S is not satisfiable

☞ HORNSAT $\in \mathbf{P}$.

5. Boolean Functions and Expressions

- An n-ary Boolean function is a mapping $\{\mathbf{true}, \mathbf{false}\}^n \to \{\mathbf{true}, \mathbf{false}\}$.

Example

The connectives $\vee$, $\wedge$, $\to$, and $\leftrightarrow$ can be viewed as binary Boolean functions and $\neg$ as a unary function.

- Any Boolean expression ϕ can be seen as an n-ary Boolean function with $n = |X(\phi)|$ by fixing the order of variables in $X(\phi)$.
- A Boolean expression ϕ with variables $x_1, \ldots, x_n$ *expresses* the n-ary function f if for any n-tuple of truth values $\mathbf{t} = (t_1, \ldots, t_n)$,

$$f(\mathbf{t}) = \begin{cases} \mathbf{true}, & \text{if } T \models \phi. \\ \mathbf{false}, & \text{if } T \not\models \phi. \end{cases}$$

where for T it holds that $T(x_i) = t_i$ for every $i = 1, \ldots, n$.

Example

$\phi = (x_1 \vee x_2) \wedge \neg x_2$ expresses the Boolean function f such that $f(\mathbf{true}, \mathbf{false}) = \mathbf{true}$ and otherwise $f(t_1, t_2) = \mathbf{false}$.

Expressing a Boolean Function as a Boolean Expression

Proposition

Any n-ary Boolean function f can be expressed as a Boolean expression ϕ_f involving variables $x_1, \ldots, x_n$.

The idea: model the rows of the truth table of f giving **true** as a disjunction of conjunctions.

- Let F be the set of all n-tuples $\mathbf{t} = (t_1, \ldots, t_n)$ with $f(\mathbf{t}) = \mathbf{true}$.
- For each $\mathbf{t} \in F$, let $D_\mathbf{t}$ be a conjunction of literals x_i if $t_i = \mathbf{true}$ and $\neg x_i$ if $t_i = \mathbf{false}$.
- Let $\phi_f = \bigvee_{\mathbf{t} \in F} D_\mathbf{t}$
- Note that ϕ_f may get big in the worst case: $O(n2^n)$.

Example.

x_1	x_2	f
0	0	0
0	1	1
1	0	1
1	1	0

$\phi_f = (\neg x_1 \wedge x_2) \vee (x_1 \wedge \neg x_2)$.

☞ Not all Boolean functions can be expressed concisely.

6. Boolean Circuits

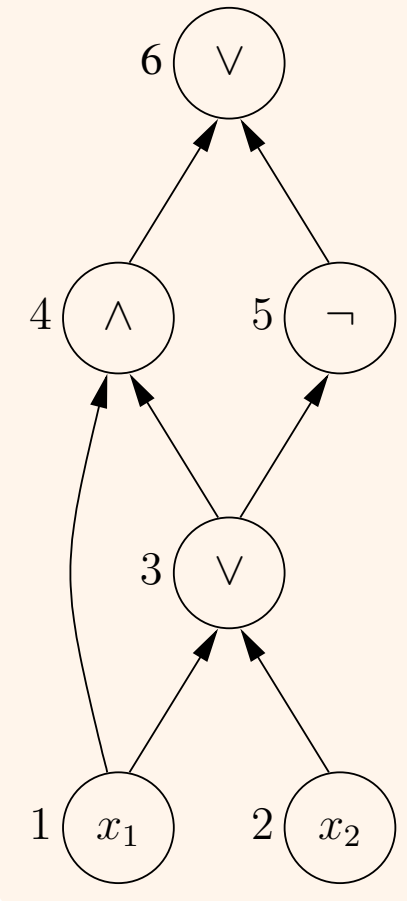
A more economical way to represent Boolean functions.

Syntax:

A Boolean circuit is a graph $C = (V, E)$ where

- $V = \{1, 2, \ldots, n\}$ is the set of *gates*,
- E must be *acyclic* ($i < j$ for all edges $(i, j) \in E$), and
- each gate i has a *sort* $s(i) \in \{x_1, x_2, \ldots\} \cup \{\mathbf{true}, \mathbf{false}, \wedge, \vee, \neg\}$ such that
  – if $s(i) \in \{x_1, x_2, \ldots\} \cup \{\mathbf{true}, \mathbf{false}\}$, the indegree of i is 0 (inputs);
  – if $s(i) = \neg$, the indegree of i is 1;
  – if $s(i) \in \{\vee, \wedge\}$, the indegree of i is 2.
- Node n is the *output* of the circuit.

Example



Semantics

A truth assignment for a circuit C is a function

$$T : X(C) \to \{\textbf{true}, \textbf{false}\}$$

where $X(C)$ is the set of variables appearing in C.

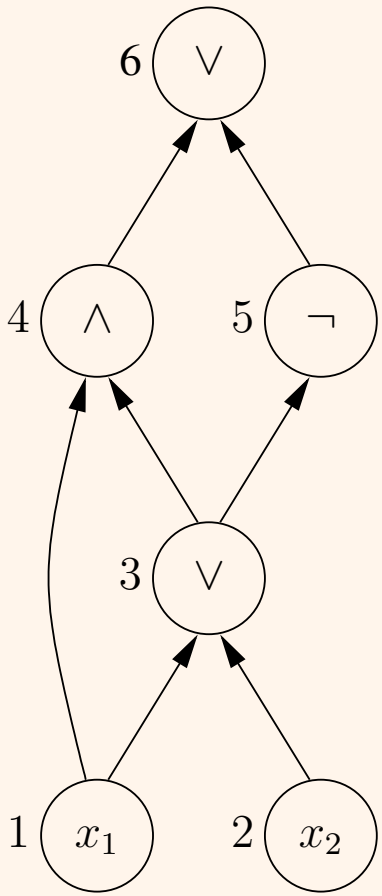
The truth value $T(i)$ for each gate i is defined inductively:

- If $s(i) = \textbf{true}$, then $T(i) = \textbf{true}$
- If $s(i) = \textbf{false}$, then $T(i) = \textbf{false}$
- If $s(i) \in X(C)$, then $T(i) = T(s(i))$
- If $s(i) = \neg$ and (j,i) is the unique edge entering i, then
  - $T(i) = \textbf{true}$ if $T(j) = \textbf{false}$, and
  - $T(i) = \textbf{false}$ otherwise
- If $s(i) = \wedge$ and $(j,i), (j',i)$ are the two edges entering i, then
  - $T(i) = \textbf{true}$ if $T(j) = \textbf{true}$ and $T(j') = \textbf{true}$, and
  - $T(i) = \textbf{false}$ otherwise
- If $s(i) = \vee$ and $(j,i), (j',i)$ are the two edgesentering i, then
  - $T(i) = \textbf{true}$ if $T(j) = \textbf{true}$ or $T(j') = \textbf{true}$, and
  - $T(i) = \textbf{false}$ otherwise

The *value of the circuit* C is $T(C) = T(n)$

Example

The circuit C:



Consider a truth assignment

$T = \{x_1 \mapsto \textbf{false}, x_2 \mapsto \textbf{false}\}$

Now

$T(1) = T(x_1) = \textbf{false}$

$T(2) = T(x_2) = \textbf{false}$

$T(3) = \textbf{false}$ (as $T(1) = \textbf{false}$, $T(2) = \textbf{false}$)

$T(4) = \textbf{false}$

$T(5) = \textbf{true}$

$T(6) = \textbf{true}$

Hence, the value of the circuit C is

$T(C) = T(6) = \textbf{true}$.

Boolean circuits vs. Boolean expressions

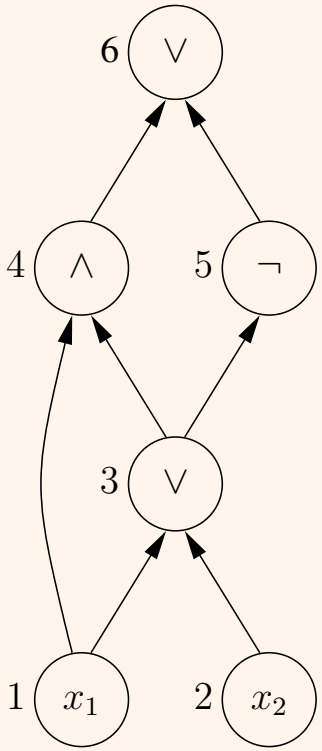
- For each Boolean circuit C, there is a corresponding Boolean expression ϕ_C.
- For each Boolean expression ϕ, there is a corresponding Boolean circuit C_ϕ such that for any T appropriate for both,

$$T(C_\phi) = \textbf{true} \text{ iff } T \models \phi.$$

  Idea: just introduce a new gate for each subexpression of ϕ, i.e., the parse tree of the expression can be seen as a Boolean circuit.
- Notice that Boolean circuits allow shared subexpressions but Boolean expressions do not.

Example

The circuit C

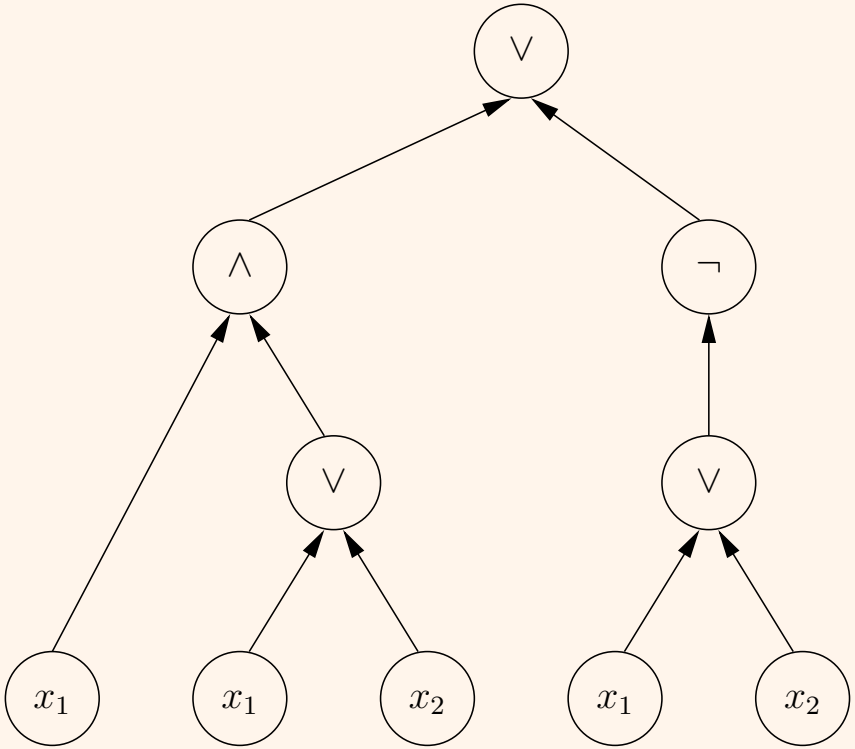


and the corresponding Boolean expression ϕ_C:

$$(x_1 \wedge (x_1 \vee x_2)) \vee \neg(x_1 \vee x_2)$$

Boolean circuit C_{ϕ_C} for ϕ_C:

$$(x_1 \wedge (x_1 \vee x_2)) \vee \neg(x_1 \vee x_2)$$



Note the difference:
in C substructure is shared

Computational problems related to Boolean circuits

Definition

CIRCUIT SAT:
INSTANCE: a circuit C.
QUESTION: is there a truth assignment $T : X(C) \to \{\mathbf{true}, \mathbf{false}\}$ such that $T(C) = \mathbf{true}$?

- CIRCUIT SAT $\in \mathbf{NP}$.

Definition

CIRCUIT VALUE:
INSTANCE: a circuit C with no variables.
QUESTION: is it the case that $T(C) = \mathbf{true}$?

- CIRCUIT VALUE $\in \mathbf{P}$.
  (No truth assignment is needed as $X(C) = \emptyset$).

Circuits computing Boolean functions

- A Boolean circuit with variables $x_1, \ldots, x_n$ *computes* an n-ary Boolean function f if for any n-tuple of truth values $\mathbf{t} = (t_1, \ldots, t_n)$, $f(\mathbf{t}) = T(C)$ where $T(x_i) = t_i$ for $i = 1, \ldots, n$.
- Any n-ary Boolean function f can be computed by a Boolean circuit involving variables $x_1, \ldots, x_n$.
- Not every Boolean function has a concise circuit computing it.

Theorem

For any $n \geq 2$ there is an n-ary Boolean function f such that no Boolean circuit with $\frac{2^n}{2n}$ or fewer gates can compute it.

However, nobody has been able to come up with a natural family of Boolean functions that require more than a linear number of gates to compute.

Learning Objectives

- The syntax and semantics of Boolean expressions — including their use in practice.
- The relationship/difference between Boolean expressions and circuits.
- Knowing the idea of representing Boolean functions in terms of Boolean expressions and circuits.
- Four computational problems related with Boolean logic and circuits: SAT, HORNSAT, CIRCUIT SAT, and CIRCUIT VALUE.